CSE 505:
Concepts of Programming Languages

Dan Grossman

Fall 2009

Lecture 8— Type Safety; Extensions to STLC

Outline

- Type-safety proof
  - Also posted in non-slide form
- Discuss the proof
  - Chart of lemma dependencies
  - Inverting multiple derivations
- Extend STλC
  (pairs, records, sums, recursion, . . .)
  - For each, sketch proof additions
  - At the end, discuss the general approach
- Not today: References, exceptions, polymorphism, lists, . . .

Review

λ-calculus with constants:

$$e ::= \lambda x.\ e \mid x \mid e\ e \mid c \qquad v ::= \lambda x.\ e \mid c$$

$$\frac{}{(\lambda x.\ e)\ v \rightarrow e[v/x]} \qquad \frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v\ e_2 \rightarrow v\ e_2'}$$

$$\frac{}{x[e/x] = e} \qquad \frac{y \neq x}{y[e/x] = y} \qquad \frac{}{c[e/x] = c}$$

$$\frac{e_1[e/x] = e_1' \quad y \neq x \quad y \notin FV(e)}{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'} \qquad \frac{e_1[e/x] = e_1' \quad e_2[e/x] = e_2'}{(e_1\ e_2)[e/x] = e_1'\ e_2'}$$

Stuck states: not values and no step applies...

Avoid stuck states to catch bugs (why would you want to get to such a state?) and make implementation easier (no need to check for being stuck)

Review Continued

Defined a type system to classify λ-terms.

Some terms have types; some don't.

$$\tau ::= \mathsf{int} \mid \tau \to \tau \qquad \Gamma ::= \cdot \mid \Gamma, x : \tau$$

$$\frac{}{\Gamma \vdash c : \mathsf{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

Theorem: A program that typechecks under $\cdot$ won't get stuck, i.e., If $\cdot \vdash e : \tau$ then e diverges or $\exists v, n$ such that $e \to^n v$.

Proof: Corollary to these lemmas:

Lemma (Preservation): If $\cdot \vdash e : \tau$ and $e \to e'$, then $\cdot \vdash e' : \tau$.

Lemma (Progress): If $\cdot \vdash e : \tau$, then e is a value or there exists an e' such that $e \to e'$.

Progress

Lemma: If $\cdot \vdash e : \tau$, then e is a value or there exists an e' such that $e \rightarrow e'$.

Proof: We first prove this lemma:

Lemma (Canonical Forms): If $\cdot \vdash v : \tau$, then:

- if τ is **int**, then v is some c
- if τ has the form $\tau_1 \rightarrow \tau_2$ then v has the form $\lambda x.\ e$.

Proof: By inspection of the form of values and typing rules.

We now prove Progress by induction on the derivation of $\cdot \vdash e : \tau$.

Progress continued

Bottom rule could conclude:

- $\cdot \vdash x : \tau$ — impossible because $\cdot \vdash e : \tau$.
- $\cdot \vdash c :$ **int** — then e is a value
- $\cdot \vdash \lambda x.\ e : \tau$ — then e is a value
- $\cdot \vdash e_1\ e_2 : \tau$ — By induction either e_1 is some v_1 or can become some e_1'. If it becomes e_1', then $e_1\ e_2 \rightarrow e_1'\ e_2$. Else by induction either e_2 is some v_2 or can become some e_2'. If it becomes e_2', then $v_1\ e_2 \rightarrow v_1\ e_2'$. Else e is $v_1\ v_2$. *Inverting the assumed typing derivation* ensures $\cdot \vdash v_1 : \tau' \rightarrow \tau$ for some τ'. So Canonical Forms ensures v_1 has the form $\lambda x.\ e'$. So $v_1\ v_2 \rightarrow e'[v_2/x]$.

Note: If we add $+$, we need the other part of Canonical Forms.

Preservation

Lemma (Preservation): If $\cdot \vdash e : \tau$ and $e \rightarrow e'$, then $\cdot \vdash e' : \tau$.

Proof: By induction on (height of) the derivation of $\cdot \vdash e : \tau$.
Bottom rule could conclude:

- $\cdot \vdash x : \tau$ — actually, it can't; $\cdot(x)$ doesn't exist.
- $\cdot \vdash c : \textbf{int}$ — then $e \rightarrow e'$ is impossible, so lemma holds *vacuously*.
- $\cdot \vdash \lambda x.\ e : \tau$ — then $e \rightarrow e'$ is impossible, so lemma holds *vacuously*.
- $\cdot \vdash e_1\ e_2 : \tau$ — Then we know $\cdot \vdash e_1 : \tau' \rightarrow \tau$ and $\cdot \vdash e_2 : \tau'$ for some τ'. There are 3 ways to derive $e_1\ e_2 \rightarrow e'$...

Preservation, app case

We have: $\cdot \vdash e_1 : \tau' \rightarrow \tau$, $\cdot \vdash e_2 : \tau'$, and $e_1\ e_2 \rightarrow e'$. We need: $\cdot \vdash e' : \tau$. The derivation of $e_1\ e_2 \rightarrow e'$ ensures 1 of these:

- e' is $e_1'\ e_2$ and $e_1 \rightarrow e_1'$: So with $\cdot \vdash e_1 : \tau' \rightarrow \tau$ and induction, $\cdot \vdash e_1' : \tau' \rightarrow \tau$. So with $\cdot \vdash e_2 : \tau'$ we can derive $\cdot \vdash e_1'\ e_2 : \tau$.
- e' is $e_1\ e_2'$ and $e_2 \rightarrow e_2'$: So with $\cdot \vdash e_2 : \tau'$ and induction, $\cdot \vdash e_2' : \tau'$. So with $\cdot \vdash e_1 : \tau' \rightarrow \tau$ we can derive $\cdot \vdash e_1\ e_2' : \tau$.
- e_1 is some $\lambda x.\ e_3$ and e_2 is some v and e' is $e_3[v/x]$...

App case, β case

Because $\cdot \vdash \lambda x.\ e_3 : \tau' \to \tau$, we know $\cdot, x{:}\tau' \vdash e_3 : \tau$. So with $\cdot, x{:}\tau' \vdash e_3 : \tau$ and $\cdot \vdash e_2 : \tau'$, we need $\cdot \vdash e_3[v/x] : \tau$.

The Substitution Lemma proves a strengthened result (must be stronger to prove the lemma)

Lemma (Substitution): If $\Gamma, x{:}\tau' \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau'$, then $\Gamma \vdash e_1[e_2/x] : \tau$.

Proof: By induction on derivation of $\Gamma, x{:}\tau' \vdash e_1 : \tau$.

Proving Substitution

Bottom rule of $\Gamma, x{:}\tau' \vdash e_1 : \tau$ could conclude (page 1 of 2):

- $\Gamma, x{:}\tau' \vdash c : \textbf{int}$. Then $c[e_2/x] = c$ and $\Gamma \vdash c : \textbf{int}$.
- $\Gamma, x{:}\tau' \vdash y : (\Gamma, x{:}\tau')(y)$. Either $y = x$ or $y \neq x$.
  If $y = x$, then $(\Gamma, x{:}\tau')(x)$ is τ' and $x[e_2/x]$ is e_2.
  So $\Gamma \vdash e_2 : \tau'$ satisfies the lemma.
  If $y \neq x$ then $(\Gamma, x{:}\tau')(y)$ is $\Gamma(y)$ and $y[e_2/x]$ is y.
  So we can derive $\Gamma \vdash y : \Gamma(y)$.
- $\Gamma, x{:}\tau' \vdash e_a\ e_b : \tau$. Then for some τ_a and τ_b,
  $\Gamma, x{:}\tau' \vdash e_a : \tau_a$ and $\Gamma, x{:}\tau' \vdash e_b : \tau_b$.
  So by induction $\Gamma \vdash e_a[e_2/x] : \tau_a$ and $\Gamma \vdash e_b[e_2/x] : \tau_b$.
  So we can derive $\Gamma \vdash e_a[e_2/x]\ e_b[e_2/x] : \tau$.
  And $(e_a\ e_b)[e_2/x]$ is $e_a[e_2/x]\ e_b[e_2/x]$.

Proving Substitution Cont'd

- $\Gamma, x{:}\tau' \vdash \lambda y.\ e_a : \tau$. (We can assume $y \neq x$ and $y \notin \mathbf{Dom}(\Gamma)$.) Then for some τ_a and τ_b, $\Gamma, x{:}\tau', y{:}\tau_a \vdash e_a : \tau_b$ and τ is $\tau_a \to \tau_b$.
  By an *Exchange Lemma* $\Gamma, y{:}\tau_a, x{:}\tau' \vdash e_a : \tau_b$.
  By a *Weakening Lemma* and $\Gamma \vdash e_2 : \tau'$, we know $\Gamma, y{:}\tau_a \vdash e_2 : \tau'$.
  So by induction (using $\Gamma, y{:}\tau_a$ for Γ (!!)), $\Gamma, y{:}\tau_a \vdash e_a[e_2/x] : \tau_b$.
  So we can derive $\Gamma \vdash \lambda y.\ e_a[e_2/x] : \tau_a \to \tau_b$.
  And $(\lambda y.\ e_a)[e_2/x]$ is $\lambda y.\ (e_a[e_2/x])$.

Exchange: If $\Gamma, x{:}\tau_1, y{:}\tau_2 \vdash e : \tau$, then $\Gamma, y{:}\tau_2, x{:}\tau_1 \vdash e : \tau$.

Weakening: If $\Gamma \vdash e : \tau$ and $x \notin \mathbf{Dom}(\Gamma)$, then $\Gamma, x{:}\tau' \vdash e : \tau$.

Lemma dependencies

- Safety (evaluation never gets stuck)
  - Preservation (to stay well-typed)
    - Substitution (β-reduction stays well-typed)
      - Weakening (substituting under nested λs well-typed)
      - Exchange (technical point)
  - Progress (well-typed not stuck yet)
    - Canonical Forms (primitive reductions apply)

Comments:

- Substitution strengthened to open terms for the proof
- When we add heaps, Preservation will use Weakening directly

Summary

What may seem a weird lemma pile is a powerful recipe:

Soundness: We don't get stuck because our induction hypothesis (typing) holds (Preservation) and stuck terms aren't well typed (contrapositive of Progress).

Preservation holds by induction on typing (replace subterms with same type) and Substitution (for β-reduction). Substitution must work over open terms and requires Weakening and Exchange.

Progress holds by induction on expressions (or typing) because either a subexpression progresses or we can make a *primitive reduction* (using Canonical Forms).

Induction on derivations – Another Look

The app cases are really elegant and worth mastering: $e = e_1\ e_2$. For Preservation, lemma assumes $\cdot \vdash e_1\ e_2 : \tau$.

Inverting the typing derivation ensures it has the form:

$$\frac{\dfrac{\mathcal{D}_1}{\cdot \vdash e_1 : \tau' \rightarrow \tau} \qquad \dfrac{\mathcal{D}_2}{\cdot \vdash e_2 : \tau'}}{\cdot \vdash e_1\ e_2 : \tau}$$

1 Preservation subcase: If $e_1\ e_2 \rightarrow e_1'\ e_2$, inverting that derivation means:

$$\frac{\dfrac{\mathcal{D}}{e_1 \rightarrow e_1'}}{e_1\ e_2 \rightarrow e_1'\ e_2}$$

continued...

The inductive hypothesis means there is a derivation of this form:

$$\frac{\mathcal{D}_3}{\cdot \vdash e_1' : \tau' \to \tau}$$

So a derivation of this form exists:

$$\frac{\dfrac{\mathcal{D}_3}{\cdot \vdash e_1' : \tau' \to \tau} \qquad \dfrac{\mathcal{D}_2}{\cdot \vdash e_2 : \tau'}}{\cdot \vdash e_1' \; e_2 : \tau}$$

(The app case of the Substitution Lemma is similar but we use induction twice at once to get the new derivation)

Adding Stuff

- Extend the syntax
- Extend the operational semantics
  - Derived forms (syntactic sugar) (with/without types)
  - Direct semantics
- Extend the type system
- Consider soundness (stuck states, proof changes)

Let bindings (CBV)

$$e ::= \ldots \mid \textsf{let } x = e_1 \textsf{ in } e_2$$

$$\frac{e_1 \to e_1'}{\textsf{let } x = e_1 \textsf{ in } e_2 \to \textsf{let } x = e_1' \textsf{ in } e_2}$$

$$\frac{}{\textsf{let } x = v \textsf{ in } e_2 \to e_2[v/x]} \qquad \frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \textsf{let } x = e_1 \textsf{ in } e_2 : \tau}$$

(Also need to extend definition of substitution...)

Progress: If e is a let, 1 of the 2 rules apply (using induction).

Preservation: Uses Substitution Lemma

Substitution Lemma: Uses Weakening and Exchange

Derived forms

let seems just like $\boldsymbol{\lambda}$, so can make it a derived form: **let** $\boldsymbol{x = e_1}$ **in** $\boldsymbol{e_2}$ a "macro" (derived form) $\boldsymbol{(\lambda x.\ e_2)\ e_1}$.

(Harder (?) if $\boldsymbol{\lambda}$ needs explicit type.)

Or just define the semantics to replace let with $\boldsymbol{\lambda}$:

$$\frac{}{\textbf{let } \boldsymbol{x = e_1} \textbf{ in } \boldsymbol{e_2 \rightarrow (\lambda x.\ e_2)\ e_1}}$$

These 3 semantics are *different* in the state-sequence sense ($\boldsymbol{e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n}$).

But (totally) *equivalent* and you could prove it (not hard).

Note: ML type-checks let and $\boldsymbol{\lambda}$ differently. (Later.)

Note: Don't desugar early if it hurts error messages!

More to come...

We'll continue making extensions next time.